INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

# *A Constraint-based Approach to Web Services Provisioning*

Eric Monfroy — Olivier Perrin — Christophe Ringeissen — Laurent Vigneron

**N° 7413**

Octobre 2010

Thème SYM


ISSN 0249-6399 ISRN INRIA/RR--7413--FR+ENG

INSTITUT NATIONAL
DE RECHERCHE
EN INFORMATIQUE
ET EN AUTOMATIQUE

INRIA

centre de recherche
NANCY - GRAND EST

# A Constraint-based Approach to Web Services Provisioning

Eric Monfroy*, Olivier Perrin†, Christophe Ringeissen† , Laurent Vigneron†

Thème SYM — Systèmes symboliques
Équipe-Projet Cassis

Rapport de recherche n° 7413 — Octobre 2010 — 39 pages

**Abstract:** In this paper we consider the provisioning problem of Web services. Our framework is based on the existence of an abstract composition, i.e., the way some services of different types can be combined together in order to achieve a given task. Our approach consists in instantiating this abstract representation of a composite Web service by selecting the most appropriate concrete Web services. This instantiation is based on constraint programming techniques which allows us to match the Web services according to a given request. Our proposal performs this instantiation in a distributed manner, i.e., the solvers for each service type are solving some constraints at one level, and they are forwarding the rest of the request (modified by the local solution) to the next services. When a service cannot provision part of the composition, a distributed backtrack mechanism enables to change previous solutions (i.e., provisions). A major interest of our approach is to preserve privacy: solutions are not sent to the whole composition, services know only the services to which they are connected, and parts of the request that are already solved are removed from the next requests.

We introduce a specific data structure, namely Message Treatment Structure, for modeling the problem. We show the interest of this data structure to express the general principles of our framework and the related algorithms.

**Key-words:** Web service, composition, privacy, constraint reasoning

* Universidad Técnica Federico Santa María, Valparaíso, Chile. E-mail: Eric.Monfroy@inf.utfsm.cl
† E-mail: FirstName.LastName@loria.fr

Centre de recherche INRIA Nancy – Grand Est
LORIA, Technopôle de Nancy-Brabois, Campus scientifique,
615, rue du Jardin Botanique, BP 101, 54602 Villers-Lès-Nancy
Téléphone : +33 3 83 59 30 00 — Télécopie : +33 3 83 27 83 19

# Une approche à base de contraintes pour la composition de services Web

**Résumé :** Nous considérons le problème de la composition de services Web. Notre approche est basée sur l'existence d'une composition abstraite modélisant l'utilisation conjointe de services Web de types différents pour remplir une tâche donnée. Notre idée consiste à instancier cette représentation abstraite d'un service composé en sélectionnant les services Web concrets les plus appropriés. Cette instanciation est basée sur des techniques de programmation par contraintes. Notre proposition réalise cette instantiation de manière distribuée: les solveurs pour chaque type de service prennent en charge une partie des contraintes et transmettent le reste de la requêtes aux services suivants. Lorsqu'un service ne peut pas construire une partie de la composition, un méchanisme de retour-arrière distribué permet de changer les solutions précédentes. Notre mécanisme de résolution distribuée nous semble bien adapté au problème de confidentialité: les solutions ne sont pas connues de l'ensemble de la composition et les parties de la requête déjà résolues sont supprimées au fur et à mesure.

Pour modéliser le problème, nous introduisons une structure de données spécifique pour le traitement des messages. On montre l'intérêt de cette structure de données pour exprimer les principes généraux de notre approche et les algorithmes correspondants.

**Mots-clés :** service Web, composition, raisonnement par contraintes

# 1 Introduction

Composition of Web services has been recently investigated from various points of view by developing different approaches based for instance on planning techniques, logical systems and appropriate transition systems. The control of a composition can be complex. One reason is the non-deterministic behavior of services [8]. Another reason is the possible failure of services involved in the composition. Therefore, exchanged messages are difficult to manage, since they include complex data related to different aspects: temporal, security, reliability, or presentation [14, 9]. The combination of all these aspects can generate a very complex design, and the resulting code can be difficult to write, to maintain, and to adapt. Implementing a composition requires to take into account different aspects such as control flow, data flow, security and reliability. Languages like WS-BPEL [1] could allow us to implement a composition covering at the same time all those aspects. However, these aspects make the task very time consuming and error prone.

Our approach is based on a distributed framework to build a composition for a given task. We use constraints to model in a declarative way various properties of services. A solver associated to each service is in charge of finding the right concrete services that are able to perform a given task with respect to various requirements expressed as constraints. At that point, a natural problem arises: how these different solvers can be combined together in order to build a composition performing this task? We address this problem in the paper by considering a simple form of composition, where (a pattern of) the composition is instantiated in an incremental way with a selection of services guided by solvers. The contribution of this paper is to present an event-based distributed framework for the composition of services. We develop the main algorithms to construct a composition thanks to solvers and backtrack mechanisms used in a distributed way. In this distributed framework, each service is building a part of the full composition. Our algorithms are expressed by using a specific data structure for the treatment of messages, called MTS for short.

The rest of the paper is organized as follows. In Section 2, we present our motivations and a case study. Section 3 presents the ingredients of our constraint-based framework. Section 4 shows how services are encapsulated into wrappers. In Section 5, we present the general principles and the algorithms used to build a composition. The solving process is discussed in Section 6. In Section 7, we show ho to simulate OWL-S constructs into our framework. The related work is discussed in Section 8 and we conclude in Section 9.

# 2 Motivations and case study

## 2.1 Motivations

The main originality of our work is the design of a distributed composition framework dealing with abstract representations of a composition of Web services.

We consider advanced composition through the use of constraints as composition requirements, at each level of the composition schema.

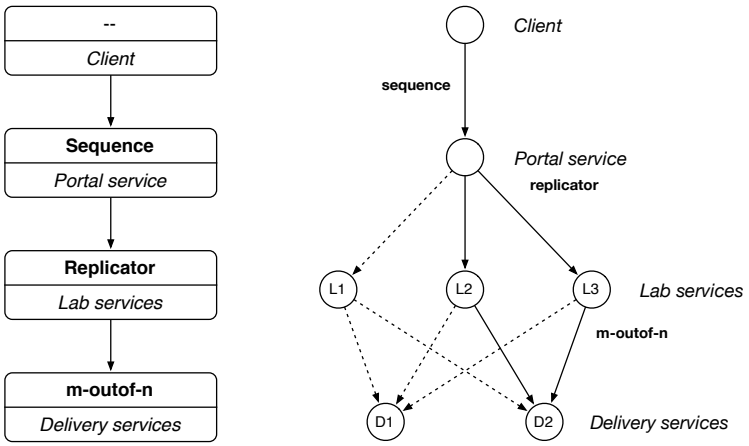


Figure 1: The process, and one possible provision.

We also minimize the information exchanged between services. This applies to the composition information, as each service has only a local view of the composition schema: by receiving some messages it will deduce which event should be fired, and therefore to which sub-services it should send follow-up messages. Minimization also applies to the knowledge handled by each service: it may do some computations involving received information, local information and public information of sub-services. This guarantees the privacy of its local knowledge, and of its performed tasks.
Another advantage of this distribution of the composition work is that constraints are solved in a distributed way and so do not depend on a unique solver.

In order to handle this dynamic composition, we do not require the design of complex services. We simply provide standard services with some wrappers (called *message treatment structures*, MTS) of operations that it can handle, each one collecting all the information for doing the selected operation and possibly by calling sub-services.
And in case of failure of a composition, a service can backtrack and select another MTS for firing another event.

Note that a detailed comparison with related work is given in Section 8.

## 2.2 Overview of our method and running case study

Let us consider a Web portal for printing numerical pictures. The components of the scenario, illustrated in Figure 1 are the client, the portal, the labs in charge of printing the pictures, and the delivery services. A printed picture has several properties, but in our example, we only retain the following properties: its size, its quality, a price.

We suppose that Web services representing the components of the Web portal exist, including several labs and delivery services. Each Web service contains the description of the operations it can achieve, the messages it can handle and properties that characterize it w.r.t. other services (Section 3.1).

The starting point is the request of the client: the client is able to express some requirements to be satisfied by the composition. For instance, a client may want to get a set of printed pictures (delivery included) in a given format, a given quality, in a given deadline, e.g.:

- print and send me these 100 pictures (no temporal or quality constraints),
- print and send me these 100 pictures at the average quality and minimum cost in the 24 hours (temporal constraint, picture quality constraints, and cost constraint),
- print and send me these 100 pictures at the $6 \times 7$ format and the $8,5 \times 11$ format as soon as possible (temporal constraint, picture format constraints).

With our method, the choreography, that is the composition of Web services for solving a request, is not known in advance. It is modeled as an abstract composition of services, through the design of composition possibilities modeled by a Message Treatment Structure (MTS, Section 4.5). A MTS corresponds to one action of a type of Web service, for example the receiving of a print estimation request by the portal from the client, and as a consequence the sending of estimation requests to several labs. Each MTS contains application constraints, and propagates the received constraints that do not concern the service playing it.
This abstract composition (i.e. non-instantiated) is described on Figure 1 (left). Using the *sequence* connector, we specify that the portal will be executed once the request of the client is sent. The portal service will replicate the request to a set of lab services (*replicator* connector), each lab having its own properties. At this level, each solver will be activated in order to select the right lab instances according to the client request. Then, we use a *m-outof-n* connector to specify that the delivery will be done by only a subset of the delivery services.

The effective choreography is the result of a dynamic combination of MTS, according to messages sent between services and also to the success or the failure of operations requested to services (Section 5).
In Figure 1 (right), we can see a possible provision of this composition. The solution returned by the solvers associated with the services instances is made of services L2 and L3 for lab services, and of service D2 for delivery.

The provisioning problem we are interested in involves multiple aspects. First, one has to specify how different types of services can be composed, i.e., what are the building blocks that are needed to orchestrate these service types in order to fulfill the user needs. This means that the ordering as well as the service types (portal, photo lab, delivery) are known. We call these building blocks the connectors, and the result is called an abstract composition.

The second aspect is to dynamically instantiate the previous abstract composition with concrete instances of services. The choice will depend on the requirements defined in the user request, but also on the possible links existing between concrete instances of services. This means, given all the constraints of the instances of services and the client request, our objective is to find one (or more) topology that satisfies the request and the constraints specified in the chosen set of services. Our proposal will achieve this objective in a distributed manner, i.e., the solvers (that equip each service) solve some constraints at one

level, and they forward the rest of the request (simplified by the local solution) to the next services. This approach preserves privacy: solutions are not sent to the whole composition, services know only the services to which they are directly connected, and parts of the request that are already solved are removed from the next requests.

Our approach is both dynamic and scalable in the sense that there is no unique and centralized point of control, but rather a set of small solvers that are in charge of instantiating the abstract composition. If one solver does not find any solution, the backtrack mechanism allows us to try another solution. Then, it also scales up since it is easy to dynamically modify, add, or remove instances of services to accommodate a high number of requests, using the algorithm we provide in Section 5.

# 3 Concepts

## 3.1 Services and service types

We consider here standard services that are defined by the set of operations they can achieve, the messages they can handle and understand, and their properties.

**Definition 1 (Service)** *A service $s$ is defined as a tuple $(n, o, m, p)$ such that*

- *$n$ is the name of $s$,*
- *$o$ is the set of operations the service is able to achieve.*
- *$m$ is the set of messages the service is able to handle.*
- *$p$ is a set of properties, i.e., some attributes of $s$*

We use the notation $s.n$, $s.o$, $s.m$, and $s.p$ to access the name of a service $s$, the set of operations it can achieve, the messages it can handle, and its properties respectively.

We do not fix here the notion of property, so usual properties can be $version$, $cost \ldots$ while more specific properties are $format$ (the format a lab can print) or $estimate$ (how much charge a lab for making an estimate).

We consider a relation that is not necessary a one to one mapping between operations and messages to request these operations since 1) a message can request several operations, and 2) a service may need several messages before achieving an operation.

We type or classify services with respect to their operations, messages, and properties. A service type $t$ is uniquely defined by a set of messages $t.m$, a set of operations $t.o$, and a set of properties $t.p$. A service $s$ is of type $t$ if $s.o \subseteq t.o \wedge s.m \subseteq t.m \wedge s.p \subseteq t.p$. Thus, a service may be of several types.

A *service variable* can be instantiated by any service. Consider a service type $t$. Then, a service variable of type $t$ can only be instantiated by a service of type $t$.

## 3.2 Web service compositions

Since services are entities that are designed to support communication, they need to be connected. To achieve this, we consider a single **directional** construct which links a service to many other ones (possibly one). We denote this construct $V(V_1, \ldots, V_n)$ meaning that the service or service variable $V$ is connected to the service or service variables $V_1$, ..., $V_n$.

Our construct does not involve any protocol (e.g., waiting for an answer, waiting for $n$ answers,...). However, using this construct, the notion of MTS (see Section 4.5) and wrapper, we can simulate and build control constructs such as the ones of OWL-S (see Section 7 for details).

Several connections can be established between the same services. For instance, considering $s_1(s_2, s_3)$ and $s_1(s_2, s_4)$, $s_1$ is connected twice to $s_2$.

### 3.2.1 Composition Patterns

A composition pattern defines a pattern of services using constructs and (typed) service variables and/or services.

**Definition 2 (Composition pattern)** *Let $\mathcal{S}$ be a set of (typed) services and let $\mathcal{V}$ be a set of (typed) service variables.*

- *a service $s \in \mathcal{S}$ or a service variable $V \in \mathcal{V}$ is called a* basic composition pattern*,*
- *given basic composition patterns $V, V_1, \ldots, V_n$, we call $V(V_1, ..., V_n)$ an* elementary composition pattern*, where $V$ is the* employer *of the elementary composition pattern, and $V_1, \ldots, V_n$ are the* employees *of the elementary composition pattern.*
- *a finite set of basic or elementary composition patterns is called a* composition pattern*.*

We say that a composition pattern is **closed** (or **instantiated**) when it does not contain any service variable. A composition pattern is often written as a (finite) list of elementary composition patterns.

**Example 1 (Composition pattern)** *$s(V_t, s_1), s_1(W)$ where $s, s_1$ are services and $V_t$ is a service variable of type $t$, and $W$ is a composition pattern variable.*

### 3.2.2 Constrained Composition Pattern

In the following, we extend the notion of pattern to constrained composition pattern. A *Constrained Composition Pattern* describes the composition of some connected services together with some information required for the composition and its behavior. To achieve this, a constrained composition pattern *ccp* defines a pattern of services using the composition pattern, services, and service variables (typed or not), together with constraints over these variables.

**Definition 3 (Constrained Composition Pattern)** *A* Constrained Composition Pattern *$C_{CP}$ is given by:*

- *a composition pattern,*

- *a set of constraints over service variables and/or between properties of services.*

Similarly to composition patterns, we will refer to basic and elementary constrained composition patterns. Thus, we do not detail the notion of constraints here. Indeed, the type of constraints that can be used depends on the internal solvers of the services (see Section 6).

Instantiating a constrained composition pattern consists in instantiating the composition pattern with a set of services instances satisfying the set of constraints.

**Example 2 (Constrained Composition Pattern)** *Consider a service type lab, such that* $\{version, estimate\} \in lab.p$. *Consider also that the type delivery requires the property version.*

*Let* $cp = (s(l, L), L(d))$ *be a composition pattern where s is a service of type portal, l is a lab, L is a service variable, and d is a delivery, and consider the following set of constraints:*

- $c_1 = l.p.version > d.p.version$ *meaning the version of l must be newer than the version of d;*
- $c_2 = L.p \subseteq lab.p \wedge L.m \subseteq lab.m \wedge L.o \subseteq lab.o \wedge L.t \subseteq lab.t$ *(L is of type lab);*
- $c_3 = l.p.estimate < 20$
- $c_4 = l.p.version < d.p.version$ *meaning the version of l must be older than the version of d;*

*Then, cp, cp and* $c_1$, *cp and* $c_2$, *cp and* $c_1 \wedge c_3$,... *are constrained composition patterns. However, note that cp and* $c_1 \wedge c_4$ *is a composition pattern that cannot be instantiated since* $c_1 \wedge c_4$ *cannot have solution.*

## 3.3 Communication

Communication between services is realized thanks to oriented messages.

**Definition 4 (Logical and physical messages)** *Consider an elementary composition pattern* $cp = S(\ldots, S', \ldots)$. *A* ***logical message*** *for cp is a term of the form* $S \to msg\_name(D_1, \ldots, D_n) \to S'$ *where:*

- *S (resp.* $S'$*) is a service variable called the* ***sender*** *(resp. the* ***receiver****),*
- *msg_name is the type of the logical message or a variable,*
- *the* $D_i$ *are typed data or typed variables.*

*A* ***physical message*** *for cp* $s \to msg\_name(d_1, \ldots, d_n) \to s'$ *is a piece of information typed by the name of the message msg_name, containing typed data* $d_1, \ldots, d_n$, *sent by the service instance s (the* ***sender****) to the service instance* $s'$ *(****receiver****).*

Since a construct is oriented, communication is also oriented: if $s(\ldots, s', \ldots)$ belongs to a construct, then $s$ can send messages to $s'$, but $s'$ cannot send messages to $s$. To this end, $s'$ must first build a composition $s'(\ldots, s, \ldots)$. Thus, remember that several connections can be established between services, so removing one connection between two services will may be not prevent them from communicating.

By extension, we denote $s \rightarrow msg\_name(d_1, \ldots, d_n) \rightarrow s_1, \ldots, s_n$ the broadcast of physical message $msg\_name(d_1, \ldots, d_n)$ to all the services $s_i$. Similarly, we also consider broadcast for logical messages. In the following, when data of the body of the messages are not relevant we will simply write $s \rightarrow msg\_name(\_) \rightarrow s'$. $s \rightarrow msg\_name \rightarrow s'$ means that the message simply does not contain data.

In the following, we will use logical messages either to store a physical message (in that case the logical message is closed, i.e. does not contain any variable), or to specify the form of the message a service is waiting for (in that case, it may contain variables). When there is no confusion between physical and logical, we will simply talk about message.

**Example 3 (Examples of messages)** *The physical message*

$$portal \rightarrow estimate(20, 4 \times 6) \rightarrow lab_1, lab_5$$

*is a message sent by the service instance portal to the labs $lab_1$ and $lab_5$ to request them an estimate of 20 prints in format $4 \times 6$.*

*The logical message*

$$S \rightarrow estimate(20, 4 \times 6) \rightarrow L, lab_5$$

*means that the service variable $L$ and the service instance $lab_5$ may receive an estimate request (to print 20 $4 \times 6$ photos) from a service.*

# 4 Wrapper of services

In our approach, a Web service must be able to execute not only its own operations, but also a part of the choreography in which this service is involved in. Thus, to be part of our framework, a service must be able to understand and execute an Message Treatment Structure (MTS) in order to build service composition, to execute coding and decoding operations, and to send follow-up messages, and to build service composition. This requires a service to have a solver to compute possible composition w.r.t. some constraints contained in the message or in services (public and private constraints, ...). Then, a Web service must be encapsulated in a wrapper that is composed of:

- some public and private constraints attached to the service,
- the ability to understand and execute a MTS,
- the ability to execute algorithms for treating messages and events, executing MTS, performing backtracking among possible compositions, algorithm to gather various constraints, and algorithm for removing compositions (see Section 5),

- a constraint solver for locally computing compositions,
- the ability to encode and decode service names as identifiers,
- the ability of memorizing information w.r.t. some indices.

The constraints related to a service are described in subsection 4.1; MTS are described in subsection 4.5; encoding of service name in subsection 4.3, and memorization in subsection 4.4. The various algorithms are described in Section 5 and solvers in Section 6.

## 4.1 Constraining services

The constraints associated to the wrapper of a service define and restrict the service instance itself either as some attributes (closed constraints that do not contain variables) or as some relations with other services. Given $s$ a service, the public part $s.cpub$ is visible to all services whereas the private part $s.cpriv$ is only accessible to the service instance itself.

**Example 4 (Public and private constraints of services)** *Consider a lab service $l$, and two delivery services $d$ and $d'$ as described in the motivating example. Then, one can imagine the following constraints for $l$:*

- *public constraint: $link(l, V) \wedge V.p.version < l.p.version$ meaning that $l$ does not (or cannot) work with a service which has a higher version that its own. Note the difference between this example and $P$ and $c_4$ from examples in Section 3.2.2: here, it is the service that impose the restriction about its use while in Section 3.2.2, it is the designer's decision.*
- *private constraints: $l.cpriv = no\_link(l, d) \wedge no\_link(l, d')$, meaning that $l$ does not want to connect and to work with services $d$ and $d'$.*

## 4.2 Events

A service can react either to one message, or to several messages coming from various services. To this end, we define the notion of events.

**Definition 5 (Event)** *Consider an employee service $S$ and $n$ employers services $S_1, \ldots, S_n$ of $S$. A logical **event** for $S$ is a set of logical messages $\{S_1 \rightarrow msg_1 \rightarrow S, \ldots, S_n \rightarrow msg_n \rightarrow S\}$. A physical event is defined similarly as a set of physical messages.*

*A physical or logical event $e = \{S_1 \rightarrow msg_1 \rightarrow S, \ldots, S_n \rightarrow msg_n \rightarrow S\}$ is fulfilled when an instance $s$ of $S$ receives $n$ physical messages that match $e$; in that case, the $S_i$ (respectively the $msg\_i$) are instantiated by the service instances that sent the messages (respectively by the body of the $msg\_i$).*

**Example 5 (Events)** *Let us consider two composition patterns $S_1(..., S, ...)$ and $S_2(..., S, ...)$. Then, $\{S_1 \rightarrow msg_1 \rightarrow S, S_2 \rightarrow msg_2 \rightarrow S\}$ is an event for $S$.*

In the following, events will be used to trigger algorithms: consider a service $s$ that can execute an algorithm $A$ associated to an event $e = \{m_1, \ldots, m_n\}$; $s$ receives messages; when $n$ messages will match $m_1$, ..., $m_n$ respectively, $e$ will

be instantiated; $s$ can raise this event and consume the messages $m_1, \ldots, m_n$; the event will trigger $A$; $A$ will execute and the event is consumed; the instantiation of $e$ can then be used as a normal term, and thus as a parameter when calling another algorithm. A raised event can be seen as a complex physical message the service sends to itself and to which it can react.

**Remark 1 (Unicity of events and messages)** *We consider that every event and message is tagged by a date. Thus, two events are never equal since they cannot be raised at the same time. This will be very important in the following since events will be used to build composition. Consider two events $e_1$ and $e_2$ for $s$, having the same structure (made of similar messages). Then, the service $s$ will be able to build a composition $sc_1$ to treat $e_1$ and a different composition $sc_2$ to treat $e_2$: even if the 2 compositions have the same pattern and the same services, they will use different instances of the services. This will enable to safely remove service instances and patterns related to an event, without damaging the composition built for other event.*

## 4.3 Coding services

Coding services names is useful to associate a service (possibly a list) with an identifier. The identifier of services $s1, ..., s_n$ is created by a service $s$, and thus only $s$ knows which services (here $s1, ..., s_n$) are identified by this identifier. Note that this identifier is unique inside $s$ in order to avoid ambiguities.

The objective of using identifiers is to ensure privacy between services since a service only needs to know the name of the service that requests it a task, and not the name of the services that it will use to achieve some sub-tasks to complete the task: a service just knows the local composition around itself.

In order to manipulate identifiers, we provide two functions:

- $identifier \leftarrow code(list\_of\_service\_names)$
- $list\_of\_service\_names \leftarrow decode(identifier)$

**Example 6 (Hiding a service name which is required later)** *Let us consider the portal $p$ of our example. It will receive estimates $e_1, \ldots, e_n$ from $n$ labs $l_i$. It will forward to the client $c$ one message containing the list of estimates $(\{(e_1, id_1), \ldots, (e_n, id_n)\})$, each estimate $e_i$ associated to the identifier $id_i$ of $l_i$ computed by $p$. Hence, the client cannot know from which labs are coming the estimates. $c$ will then send a response with the estimate $e_i$ it will select, together with the identifier $id_i$. That way, $p$ will be able to know from which lab was the estimate, and thus will send it a printing request.*

## 4.4 Memorizing information

In our framework, memorization will be necessary in the algorithms of the wrapper to help backtracking: current service composition, possible compositions, current MTS, possible MTS, messages that implied follow-up messages, ... will thus be stored. This is done with two operators, *get* and *put*, where the memorized data is attached to one/several indices:

- $put(ind_1, \ldots, ind_n, D)$ to store data $D$ indexed by $ind_1, \ldots, ind_n$

- $D \leftarrow get(ind_1, \ldots, ind_n)$ to retrieve data $D$ knowing $ind_1, \ldots, ind_n$

**Example 7 (Memorizing information)** *We give here an example used in Algorithm 5.3. $put(current\_set\_sc, se, set\_SC)$ stores the remaining possible service compositions $set\_SC$, associated with two indices: $current\_set\_sc$ to name the stored information, and the event $e$ for which the possible service compositions of $set\_SC$ were computed*[1]*.*

*In Algorithm 5.6.2, the wrapper get back the possible compositions in order to backtrack: $set\_SC = get(current\_set\_sc, e)$. Using the index $current\_set\_sc$ we retrieve the type of data we are looking for; and using the second argument, we are sure to get the remaining possible compositions computed to treat the event $e$, since the service could be treating several events at the same time.*

## 4.5 MTS: Message Treatment Structure

The choreography of all the services needed to achieve the objective is done thanks to a set of *Message Treatment Structure* (MTS). Globally, these MTS define the behavior (messages, communication, operations, and composition) of the services in order to achieve the objective (e.g., printing photos through a photo portal).

Each MTS defines the steps a service must locally execute when instantiating a given event: how to build the needed service composition, which follow-up messages the service must send to other services of the composition in order to distribute sub-tasks and to achieve the global task (although the service does not know itself about the global task, it knows its own role in it), the coding and decoding of services the service must do in order to carry on the task later, the operation the service must execute, ...

A MTS wraps a service operation *op* and defines the behavior of this service w.r.t. other services in order to perform this operation *op* inside a global task. When instantiating an event, a service will retrieve and select one of the corresponding MTS and use it to know what it has to do and how it has to communicate and cooperate with other services. Note that several MTS can match an event as it exists many ways to achieve the same task. Thus, it is the responsibility of the service to select one of the possible MTS (and to change of MTS in case it cannot complete the task with the selected one).

Whatever service able to execute an MTS can be part of our framework. MTS can be added without modifying services: thus, new composition and task can be achieved with the same services without modifying them. Services can be added without modifying MTS: thus, more services can be available to execute the same tasks. If a service provides a new action, this action will be available in our framework only if a MTS is created to encapsulate it.

### 4.5.1 Message Treatment Structure: Definition

**Definition 6 (Message Treatment Structure)** *A MTS is composed of:*

[1] Several indices are needed for the following reason. Each time, the first index will be the name (e.g., *current_set_sc*) of the information which is stored; the second argument will be a logical message or an event (treated as a syntactic term) since a service can be treating several tasks at a time; thus, the first argument would not be sufficient to determined the required data: *put* would erase still needed data (writing on some data with the same index) or *get* would have several possibilities (considering 2 *put* with the same index who stored 2 data).

- *An event e equal to*

  $\{S_1 \to msg_1(D_{1,1}, \ldots, D_{1,l_1}) \to S, \ldots, S_n \to msg_n(D_{n,1}, \ldots, D_{n,l_n}) \to S\}$

  *which is the index to access this MTS. When messages are responses to messages from S (i.e., S sent a message to $S_1, \ldots, S_n$ and it is waiting for a response), then S can consider a time limit for receiving the responses (see Example of MTS 4.6.4).*

- *A message selection number select, together with a message selection criterion criterion: this attribute allows for defining the number of messages the MTS is using (select $= 1, all$, or $m$) among the n messages of e. When select $= 1$ or $m$, criterion can be $first$ (the first message), indeterminism (one of the message), bestmatched (the message that optimizes a function),...*

  *Note that all the message of the event are consumed, even if the MTS later uses only some of them.When e consists of only one message, select must be set to 1 and criterion is not relevant.*

- *A constraint extraction function* constraint_of*: this function extracts the constraints from the messages of the event; this function enables the service to retrieve some constraints that could be implied by the messages or included in the messages;*

- *A condition switch: this is a Boolean condition over the data of the messages that are selected among the n messages of the event e. Formally, switch is a first-order formula built over the set D of variables in the event, where $D = \bigcup_{i=1}^{n}(\bigcup_{j=1}^{l_i} D_{i,j})$. Variables in D are assigned when the event is raised, and switch is a closed formula that is evaluated to $true$ or $false$. When switch is true, the then part of the MTS will be executed; otherwise the else part.*

- then *part consists of:*

  - *a constraint composition pattern given by $CP$ (see 3.2.1) and $C_{CP}$ (see 3.2.2): they define the constraint composition pattern that will be used by the service in order to build/reuse a new part of the composition w.r.t. this MTS; this corresponds to the local part of the global composition the service will know of.*
  - *decodings: a list of decoding and coding operations to retrieve service names from some identifiers that appear in a message data, and to code service names that may be used in the operation (see MTS 4.6.4).*
  - *op: the internal operation the service must execute when receiving the messages associated to the MTS;*
  - *codings: a list of coding operations to encode service names as identifiers;*
  - *a set of follow-up messages of the form $s \to msg\_name(\_) \to s'$, i.e., a message together with the names of the services (or variables that will be instantiated before sending the message) to which ($\to s'$) and from which ($s \to$) the message is sent.*

- else *part (possibly empty). This part is structured as the* then *part.*

### 4.5.2 Use of MTS

When receiving a message, a service $s$ will verify whether this message can immediately participate in order to instantiate an event of one of its MTS. $s$ then selects the messages of the event it will use using *select*. Using the constrained composition pattern defined in the MTS, together with messages and services constraints, $s$ will define in which (or for which) service composition $cp = s(s_1, \ldots, s_l)$ it will achieve its task. If the given condition *switch* is true, the service will execute the *then* part of the MTS, otherwise, the *else* part. Executing one of these part will consist in executing de/codings operations (*decodings*), executing the main operation (*op*), executing coding operations (*codings*) and finally sending some follow-up messages to the services $s_1$, ..., $s_l$ of *cp*.

Since the event can match several MTS, and that the composition pattern of each MTS can be instantiated by various services, the service may also need to backtrack and to try other composition when the computed/selected one cannot achieve the required sub-task.

### 4.5.3 MTS and complex composition patterns

A MTS can only define the "construction" of an elementary composition pattern, i.e., one single construct $s(s_1, \ldots, s_n)$ where the $s_i$ are services or services variables. Thus, for treating or building a composition pattern which is not elementary, several MTS are required. Although one could find that this is a limitation, this has the advantage of keeping privacy up to one service view (i.e., one construct); if one is less concerned with privacy, one can communicate service names in messages between several levels. Moreover, this enables us to be closer to operators of OWL-S for instance (see Section 7).

### 4.5.4 Variables of MTS

Variables appearing inside a MTS are linked, and are global to the MTS. Thus, a service variable $L$ that appears in *mts.decodings* will represent the same object that the variable $L$ appearing in *mts.codings* for example. If $L$ is instantiated to $s$ in *decodings* (e.g., using a decoding operation), then $L$ is also instantiated to $s$ if it appears in *codings*. To make clear this mechanism, we will use the notion and notation of context: $fup|_{codings}$ will denote the sending of follow-up messages in the context of *codings* (i.e., some service variables used in $fup$ maybe instantiated by some operations of *codings*). This is also valid for a solution *sc* of service composition (i.e., service instantiation computed by the solver): $codings|_{sc}$ denotes the *codings* operation in which some variables maybe instantiated by the solution *sc*. In case a set of commands is modified by several context, we will note for example: $codings|_{decodings,sc}$. The order for executing command is thus important; commands of *codings* could not benefit from the context of *op* commands if these ones have not been executed before. This will become clear in the description of the *execute_mts* algorithm (see 5.4).

## 4.6 Examples of MTS

### 4.6.1 A portal $p$ receives a request for estimates from a client $c$

This MTS is simple. The client sends a request for estimates given a format and a number of photos. The MTS associated with the portal $p$ will use the constraints specified within the message to select some labs (using *constraint_of* to extract the parameters). The designer of the MTS also specifies that the portal will not work with the lab "Kodji". Then, when the lab instances are instantiated, it will follow-up the message to the labs, with an encoded id of the client.

$event = c \rightarrow estimate(20, 4 \times 6) \rightarrow p$

- *select* = 1: only one message compose the event, thus *select* must be set to 1 and there is no criterion *criterion*.
- $constraint\_of = (l_i.p.minprinting < 20 \wedge l_i.p.format = 4 \times 6)$
- *switch* = *true*: there is no condition
- *then* part:
  - $CP = p(L)$: the portal $p$ will ask for estimate to labs (to be instantiated)
  - $C_{CP} = \forall l_i \in L, l_i.p.name \neq "Kodji"$
  - decodings = $\emptyset$
  - operation = $\emptyset$
  - codings = $id \leftarrow code(c)$
  - fup = $p \rightarrow estimate(20, 4 \times 6, id) \rightarrow l_i$
- *else* part = $\emptyset$

### 4.6.2 A client $c$ receives a list of estimates from a portal $p$

This MTS is simple. The client receives a list of estimates from a portal (together with identifiers to hide the name of the labs). $c$ chooses one of the estimate and sends back a printing request to the portal.

As the event is composed of one message, the selection is direct. Moreover, the switch is always true and there is no *else* part in the MTS.

$event = p \rightarrow list\_of\_estimates(\{d_1, id_1\}, ..., \{d_n, id_n\}) \rightarrow c$

- *select* = 1: only one message compose the event, thus *select* must be set to 1 and there is no criterion *criterion*.
- *switch* = *true*: there is no condition
- *then* part:
  - $CP = c(p)$: client $c$ will then send a printing request to the portal $p$
  - $C_{CP} = true$
  - decodings = $\emptyset$

- operation = choice of an estimate $d_i$ among $\{d_1, , \ldots, d_n\}$
- codings = $\emptyset$
- fup = $c \rightarrow print\_to\_portal(d_i, id_i) \rightarrow p$

- *else* part = $\emptyset$

#### 4.6.3 A portal p receives a printing request from a client $c$

This MTS is also simple. The event is a single message sent by $c$; this message contains the chosen estimate and the identifier of the lab that provided the estimate (the client can not know from which labs the estimates come from). Thus $p$ has to decode this identifier and to forward the printing request to the "coded" lab.

$event = c \rightarrow print\_to\_portal(d, id) \rightarrow p$

- $select = 1$: thus, there is no criterion.
- $switch = true$: there is no condition
- *then* part:
  - $CP = p(L)$
  - $C_{CP} = true$
  - decodings = $L \leftarrow decode(id)$
    where $id$ is the lab identifier created by the portal $p$ when it received the estimate from the lab. $id$ is only understandable by $p$.
  - operation = forward printing request (the estimate $d$) to the corresponding lab.
  - codings = $\emptyset$
  - fup = $print\_to\_portal(d) \rightarrow L$
- *else* part = $\emptyset$

#### 4.6.4 A portal $p$ receives some estimates from some labs

The portal $p$ sends an estimate request to $n$ labs, and waits for at least $m$ positive answers. Assume $p$ receives $k$ estimates from $n$ labs. If $k \geq m$, $p$ will select $m$ of the best estimates (w.r.t. a $bestfit$ in the $criterion$); otherwise, $p$ considers that the choice is too poor to make a good proposition to the client; it will thus forward to the client a negative answer.

Note that $id$, in the received estimates, corresponds to the identifier of the client (the labs can not know who is the client).

$event = l_1 \rightarrow estimate(d_1, id) \rightarrow p, ..., l_k \rightarrow estimate(d_k, id) \rightarrow p$ $(m \leq k \leq n)$

- $select = m$: $m$ out of the $k$ messages are selected with $criterion = bestfit$. Note that *select* will fail if $m > k$
- $switch = (m \leq k)$: the switch is true only if more that $m$ answers were received.

- *then* part: estimates are forwarded to the client
  - $CP = p(c)$: the portal will send a resume of the estimates to the client.
  - $C_{CP} = true$
  - decodings:
    - $c \leftarrow decode(id)$
      Meaning: $p$ decodes the identifier it had created (when receiving the estimate requested from the client; thus it has sent to the labs an identifier they could not interpret, and thus they could not know which client it was.
    - for all $i$: $id_i \leftarrow code(l_i)$
      Meaning: $p$ encodes the names of the labs so the client does not know which labs are involved.
  - operation = collect the estimates $d_i$ from identifier lab $id_i$ in order to forward a list of couples (estimate, lab identifier) $\{d_1, id_1\}, ..., \{d_k, id_k\}$ to the client $c$
  - codings = $\emptyset$
  - fup = $p \rightarrow list\_of\_estimate(\{d_1, id_1\}, ..., \{d_k, id_k\}) \rightarrow c$
- *else* part:
  - $CP = p(c)$: the portal will send a resume of the estimates to the client.
  - $C_{CP} = true$
  - decodings = $c \leftarrow decode(id)$ (same meaning as the *then* part).
  - operation = $\emptyset$
  - codings = $\emptyset$
  - fup = $p \rightarrow no\_valid\_estimate(sorry) \rightarrow c$
    In that case, the portal did not receive enough estimates to make a good proposition to the client.

# 5 Principles and Algorithms

This section describes the various mechanisms used for treating messages and building new parts of service composition in order to achieve some tasks.

## 5.1 Overall description

### 5.1.1 Problem to solve

Given a new message that participates for fulfilling an event, the provisioning problem is now to find:

- a composition pattern to handle the task implied by the event,

- an instantiated service composition such that the constraints of the event, the constraints of the composition pattern, and the constraints of the various services involved in the composition are satisfied.

Of course, the instantiation of the composition may lead to send messages to other services, and to recursively execute the fragment defined above. For instance, in the example, once the portal got the message *estimate*, it follows-up new messages to laboratories participating in the chosen composition, these ones then follows-up messages to the deliveries participating in the labs' chosen compositions, ...

### 5.1.2 General principle

Globally, achieving a task will require services to build and link new compositions to existing compositions. Some "sub"-tasks are then forwarded to the newly participating services or to already linked services. Since services decide and compute locally the newly required composition, it may happen that these new elements cannot carry on the task. It thus appears some backtrack phases to change part of the composition already built: in this case, a new composition is tried to achieve the task. The best case is when a task is achieved without backtracking: the local decisions made by the services enable to globally achieve the task. The worst case is when it is impossible to achieve a task: this means that services tried all the possibilities to construct a composition for realizing the task, but none of them were successful.

### 5.1.3 Running a service

When a service $s$ receives a message $s' \rightarrow msg \rightarrow s$ coming from a service $s'$, the service $s$ will trigger a process to verify if this message can help completing and fulfilling an event corresponding to an MTS, i.e., corresponding to a task. In the negative case, the service just carry on its current tasks (if any) and wait for a new message. In the positive case, the event is raised by using the related MTS. The MTS collects all the information to build the composition, to decode service names (decodings part of the MTS), to execute the required task, to send follow-up message containing the sub-tasks (using the Fups of the MTS), and coding service names (codings part). Basically, to execute a MTS, a service will achieve the following:

- compute a composition w.r.t. the task to execute and the various constraints (message constraints, service constraints, composition pattern constraints, ...). To this end, the service $s$ uses the information contained into the MTS, its own constraints, the public constraints of the possibly participating services, and its solver (called inside the *possible_sc* algorithm):
- execute the rest of the MTS in the context of the computed and selected service composition:
  - execute the decoding operation to get from its own memory some service names that were hidden to other services in order to keep privacy (the other services just know an identifier, but only $s$ is able to associate this identifier with a service);

- execute the operation required by the message
- execute the coding operations to memorize the association of some identifiers to some services; this enables to get private the name of the services.
- send the follow-up message and memorize the context (current service composition, event, ...) which implied sending the fup;

A composition in charge of some sub-tasks may fail. In this case, the service must try others compositions (either a different instantiation of service variables, or a different MTS related to the same event). To this end, the service has at its disposal some backtrack mechanisms: first to try a new instantiation of services keeping the same MTS; and if this does not succeed, it will try another MTS. If it cannot succeed after having tried all these possibilities, it will request the services appearing in the event to backtrack: it is not able to achieve the task, but may be another service can (i.e., the same backtrack mechanism is performed at the upper "level").

This backtrack mechanism is associated to a "cleaning" mechanism. Compositions that are not needed anymore are removed (thus, messages based on this composition cannot be sent anymore), memories are cleaned, ... Note that the backtrack and cleaning mechanisms are coupled and they call each others.

Basically, backtrack requests from a service $s$ are sent to employers to notify them that $s$ cannot manage the task anymore (and thus, cannot manage the message it received): in this case, the employers will try another composition, in which $s$ may possibly participate and possibly with the same message (but in this case, at least one of the other employee will be changed).

Stop requests from a service $s$ are sent to its employees: $s$ cannot manage anymore a task for which it has forwarded sub tasks to its employees; since $s$ stop treating this task, its employees must follow and do the same.

## 5.2 Treating messages

This first algorithm is in charge of treating arriving messages. It is triggered whenever a new message arrives. The message is stored with the set $M$ of untreated messages. All possible events composed of messages from $M$ are then considered; the first event $e$ that matches the index of an MTS is kept (note that several events could match); messages composing $e$ are then consumed and removed from the set of untreated messages; the event $e$ is raised in order to trigger the *treat_event* algorithm (see next section) and *treat_msg* terminates. In case there is no corresponding MTS, the service just carry on its tasks and waits until receiving a new message.

When *treat_msg* is triggered, the local composition around $s$ contains at least the pattern $s'(\ldots, s, \ldots)$. The service composition is not changed by *treat_msg*.

**Algorithm 1** *treat_msg* **triggering event:** $s' \to m \to s$

```
    % get the messages not yet treated by the service
    M ← get(set_untreated_msg) ∪ {s' → m → s}
    % if there is a subset of the set of untreated messages M
    % that matches the index of at least one mts
 5: if ∃e ⊆ M such that match(e, mts) then
      % messages of e are consumed
      put(set_untreated_msg, M \ e)
      % the event is raised to trigger other algorithms
      raise(e)
10: else
      put(set_untreated_msg, M)
    end if
```

## 5.3 Treating events

The *treat_event* algorithm is executed by a service $s$ when an event $e$ (different from a backtracking request or a stop request) has been raised; $e$ is also consumed and then, it can only be used as a parameter for calling other algorithm. First, $s$ stores the new event in the set of currently being treated events. It then looks for all the MTS corresponding to the event $e$ (we do not detail here the *get_all_MTS* function), and it selects one of them (*backtrack_mts*). The MTS switch is checked to know whether the *then* part or the *else* part of the MTS will be executed. The possible service compositions are then computed (*possible_sc*). Finally the operations contained in the MTS are treated (*execute_mts*). Note that all possible service compositions and MTS are stored here, whereas the currently used MTS, and the currently used service composition are stored in other algorithms.

When *treat_event* is triggered, the local composition around the service $s$ contains at least the patterns $s_1(\ldots, s, \ldots), \ldots, s_n(\ldots, s, \ldots)$ where $n$ is the number of messages in the event. After the execution of *treat_event*, the composition is eventually completed with $s(\ldots, s'_i, \ldots)$ where the $s'_i$ can be new services or services from $s_1, \ldots, s_n$ (e.g., to send an answer).

**Algorithm 2** *treat_event* **triggering event:** $e \neq s' \rightarrow bt\_request(\_) \rightarrow s$

```
    % e is added to the set of events currently being treated
    SE ← get(current_treated_events)
    put(current_treated_events, SE ∪ e)
    % get all the mts matching the event
 5: set_MTS ← get_all_MTS(e)
    put(current_set_mts, e, set_MTS)
    % select a mts among MTS
    mts ← backtrack_mts(e)
    % select the then or else part of the MTS depending on the condition switch
10: if mts.switch|_e then
      mtsp ← mts.then
    else
      mtsp ← mts.else
    end if
15: % computes all possible instantiated service compositions
    set_SC ← possible_sc(mtsp, e)
    put(current_set_sc, e, set_SC)
    sc ← backtrack_sc(e)
    execute_mts(sc, mtsp, e)
```

After *execute_mts*, at the end of the algorithm, we do not "clean" the memories and the connections (compositions used to treat the event). Indeed, it may happen that latter, one of the employees of $s$ for $e$ cannot achieve its task. In that case, $s$ will have to remember what it did for treating $e$. We only apply this cleaning mechanism when backtracking or stopping, as it is later explained in this section. One could use some time limit $t$: if $t$ seconds after terminating *execute_mts* there is no backtrack request from an employee of $s$ for the event $e$, then $s$ removes the composition it has built for $e$ and clean its memory. However, this is not necessary.

**Remark 2 (Matching of events and consuming messages)** *An event can be formed of several messages. It thus can appear that an event is included in another event. For example, the event $s' \rightarrow m \rightarrow s$ is included in the larger event $s_1 \rightarrow m \rightarrow s, \ldots, s' \rightarrow m \rightarrow s$. In that case, treat_event will be triggered with the smallest matching event.*

## 5.4 Executing commands from a MTS

The next algorithm (*execute_mts*) is in charge of carrying on the treatment of the task implied by the event. Testing the switch, de/codings, operations, follow-up messages, and coding operations are extracted from the MTS *mts* and executed in an increasing context (the event $e$ and the service composition *sc* for executing the command of *mts.decodings* and testing *mts.switch*, then *event*, *sc*, and *mts.decodings* when executing *mts.op*, and so on) since the execution of each set of operations can instantiate some service variables.

When sending the follow-up messages, $s$ stores the composition and the event that produced this message. This information will be necessary when backtracking. Note also that all the follow-up messages associated to an event

are also stored (last line of the algorithm). *decodings* enable to decode service name from an id (and thus, to instantiate service variables); the operation may also instantiate some service variable; the follow-up messages enable to forward or request sub-tasks to some services; the post-operations enable to code and "memorize" some service names.

When entering $execute\_mts$, the local composition around the service $s$ contains at least the patterns $s_1(\ldots, s, \ldots), \ldots, s_n(\ldots, s, \ldots)$ where $n$ is the number of messages contained in the event parameter $e$. Executing the MTS will eventually complete the local composition with $s(\ldots, s'_i, \ldots)$ where the $s'_i$ can be new services or services from $s_1, \ldots, s_n$ (e.g., to send an answer).

**Algorithm 3** $execute\_mts(sc, mtsp, e)$

% execute codings and decodings in the context of the current service
% composition $sc$ and $e$
$exec(mtsp.decodings|_{e,sc})$
% The context for executing the operations is growing
$exec(mtsp.op|_{e,sc,mtsp.decodings})$
% execute coding operations in the context of the service composition being built
% and previous operations
$exec(mtsp.codings|_{e,sc,mtsp.decodings,mtsp.op})$
% For each follow-up message from the list fups
**for all** $s \to Fup_i \to S_i \in mtsp.fups$ **do**
  % sends the follow-up messages $Fup_i$ in the context of $e$, $sc$, $decodings$, and $op$
  % to each service $S_i$ in the context of $sc$, $decodings$, and $op$.
  $send((s \to Fup_i \to S_i)|_{e,sc,mtsp.decodings,mtsp.op,mtsp.codings})$
  % memorize the fup together with the message $m$ being treated, the composition
  % $sc$ being built w.r.t. the part of the mts $mtsp$
  $put(sc\_implies\_fup, (s \to Fup_i \to S_i)|_{e,sc,mtsp.decodings,mtsp.op,mtsp.codings}, sc)$
  $put(event\_implies\_fup,$
  $(s \to Fup_i \to S_i)|_{e,sc,mtsp.decodings,mtsp.op,mtsp.codings}, e)$
**end for**
$put(fups\_implied\_by\_e, e, mtsp.fups|_{e,sc,mtsp.decodings,mtsp.op,mtsp.codings})$

## 5.5 Computing possible service composition given a MTS

The algorithm *possible_sc* computes all the possible service instantiations corresponding to the current MTS. First, the constraints included in the event $e$ ($constraint\_of(e)$) are solved together with the constrained composition pattern $mts.CCP$ (the composition pattern is $CP$ and the constraint over the pattern is $C_{CP}$), the public constraints of the service $s$ ($mts.cpub$), and the private constraints of the service $s$ ($s.cpriv$). This gives a first set of substitutions $Sol1 = sol_1, \ldots, sol_n$. Each substitution $sol \in Sol1$ instantiates service variables $S_1, \ldots, S_m$ by services $s_1, \ldots, s_m$, which is written $sol = \{S_1 \mapsto s_1, \ldots, S_m \mapsto s_m\}$. Each substitution $sol \in Sol1$ is such that its domain $Dom(sol) = \{S_1, \ldots, S_m\}$ is the set of service variables in $mts.CCP$.

In a second step, each substitution $sol \in Sol1$ must also satisfy the public constraints of all the services $s_i$ appearing in $sol$ ($\bigwedge_{S \in Dom(sol)} sol(S).cpub(e)$). Note that *cpub* is parametrized by the event $e$ ($cpub(e)$ with $e = \{s_1 \rightarrow m_1 \rightarrow s, \ldots, s_n \rightarrow m_n \rightarrow s\}$). For example:

- a service $s$ can decline working with a service $s'$ for a given task, but accept for another task; $e$ contains the logical messages, thus it implicitly also contains the tasks to be achieved,
- or, a service $s$ can decline working with a service $s'$ for an event $\{s \rightarrow s_i\}$, but can accept for the event $e$.

When a solution $sol \in Sol1$ satisfies the public constraints of its services, it is finally considered as a solution; note that we keep a couple made of the services instantiations ($sol$) and the instantiated composition pattern $CP|_{sol}$, where $CP$ is the composition pattern of $mts.CCP$. The list of such couples is then returned as the result of the *possible_sc* algorithm.

When executing *possible_sc*, the local composition around $s$ is of the form $s_1(\ldots, s, \ldots), \ldots, s_n(\ldots, s, \ldots)$ where $n$ is the number of messages contained in the event $e$. The composition is not changed by this algorithm.

**Algorithm 4** *possible_sc*$(mts, e)$

$(CP, C_{CP}) \leftarrow mts.CCP$
$Sol1 \leftarrow solve(mts.constraint\_of(e) \wedge C_{CP} \wedge s.cpub \wedge s.cpriv)$
% Solutions are filtered again w.r.t. the public constraints/properties
% of the services appearing in a candidate solution of $Sol1$
$Sol = \emptyset$
**for all** $sol \in Sol1$ **do**
  **if** $solve(sol \wedge C_{CP} \wedge \bigwedge_{S \in Dom(sol)} sol(S).cpub(e)) \neq \emptyset$ **then**
    $Sol \leftarrow Sol \cup \{(sol, CP|_{sol})\}$
  **end if**
**end for**
$return(Sol)$

## 5.6 Selecting and backtracking over possible service compositions

### 5.6.1 Needs for backtracking and alternatives

The mechanism must select a service composition between the various possibilities given by the different MTS and the different compositions for each MTS.

At this stage, there are two possibilities:

- compute all the solutions corresponding to all MTS and all compositions for each MTS.
- first select a MTS, and then select a composition for this MTS.

We have chosen the second possibility since it requires less computations of compositions. Note that the way MTS and composition are selected are different. On the one hand, a MTS is selected to get a composition pattern according

to some criteria that can vary from one service to another (e.g., composition pattern that requires less services, or composition pattern that reduce communication, ...). On the other hand, a composition is selected to get service instances according to a given MTS and constraints given by the constrained composition pattern, the constraints of the service, and the constraints contained in the messages of the event).

We have a backtrack mechanism with two phases. First we try to backtrack over possible compositions, and when it is no more possible, over MTS. Hence, backtracking over possible composition w.r.t. a MTS will trigger the backtrack over MTS when no more composition remains. Backtracking over MTS will require the service to stop treating the event when no more possible MTS remains; in this case, it warns the services that appear in the event so they can backtrack.

When changing a composition to perform sub-tasks (i.e., backtracking on compositions), a service $s$ will inform the services of this composition so they can stop treating these sub-tasks. These latters will also propagate this information to their "employees", and warn their other employers (the services appearing in the event they are treating, except $s$ since the stop came from it) that they stop treating these sub-tasks (see Section 5.7 for the detailed algorithms).

This section describes the algorithms related to the backtrack mechanism: *backtrack_MTS* for backtracking among MTS, and *backtrack_sc* for backtracking among service compositions. We specify also the algorithm that is fired when a service receives a backtracking event (*treat_bt_request*). When a service cannot backtrack anymore (it has unsuccessfully already explored all its possibilities in terms of MTS and compositions), it requests its "employers" (i.e., services that requested it a task that finally it cannot perform due to its incapacity to build a corresponding composition) to backtrack.

### 5.6.2 Selecting and backtracking over MTS

The *backtrack_mts* algorithm is in charge of managing the backtracks over possible MTS corresponding to an event. First, the service $s$ looks for the currently remaining (w.r.t. the event $e$ which was passed as parameter) MTS using the get command. If some MTS are remaining, one of them is selected (we can think here of some possible strategies such as to limit the number of services required by the composition). Otherwise, a backtrack request is sent to the services $s_i$, i.e., the services which sent the messages forming the event, and thus that had requested $s$ to achieve a task. Note that $s$ recalls to the $s_i$ for which message this backtrack request is sent ($s_i \rightarrow m_i \rightarrow s$ is passed as argument). $s$ stops treating the event $e$ since it could not treat correctly the demand of the $s_i$: none of the possible MTS could provide a service composition satisfying $s_i$. Then, $s$ requests its employees for $e$ to stop (through the $STOP\_current\_composition(e)$ algorithm). In fact, $s$ stops treating the event $e$.

When executing *backtrack_mts*, the local composition around $s$ contains at least the following patterns $s_1(\ldots, s, \ldots), \ldots, s_n(\ldots, s, \ldots)$, where $n$ is the size of the event $e$ and the ... can be empty. After executing this algorithm:

- in case there were some remaining possible MTS, the composition around $s$ is unchanged; it will be later changed when selecting the needed service composition,

- or (in case the service has unsuccessfully tried all its possibilities), after the call of $STOP\_current\_composition(e)$, $s$ can vanish from the patterns $s_i(\ldots, s, \ldots)$; indeed these patterns were created for a task in which $s$ cannot participate anymore. But removing $s$ from the pattern will be made by the $s_i$ when treating the backtrack request.

**Algorithm 5** *backtrack_mts*$(e)$

$set\_MTS \leftarrow get(current\_set\_mts, e)$
**if** $set\_MTS = \emptyset$ **then**
  **for all** $s_i \rightarrow m_i \rightarrow s \ \in e$ **do**
    $send(s \rightarrow \ bt\_request(s_i \rightarrow m_i \rightarrow s) \ \rightarrow s_i)$
  **end for**
  $STOP\_current\_composition(e)$
**else**
  $mts \leftarrow selectMTS(set\_MTS)$
  $put(current\_mts, e, mts)$
  $put(current\_set\_mts, e, set\_MTS \setminus \{mts\})$
  $return(mts)$
**end if**

### 5.6.3 Selecting and backtracking over service compositions

This is the second level of backtrack which consists in changing service composition given a MTS.

Given an event, the *backtrack_sc* algorithm returns a couple made of a service instantiation and the instantiated composition pattern; this couple is selected with the *select_sc* function. The composition is then established ($connect(sc)$ where $sc$ is of the form $s(\ldots)$). When there does not remain any possible instantiation ($set\_SC = \emptyset$), the algorithm calls the *backtrack_mts* algorithm to get another MTS.

If $s$ had already made a composition to treat the event $e$ (i.e., $get(current\_sc, e)$ returns a pattern different from $NULL$) it first removes this composition and requests the services of this pattern to stop since there are no employees anymore to handle $e$. Otherwise, $get(current\_sc, e)$ does not return a pattern, and no composition has already been tried to handle this event.

Consider the event parameter $e$ to be $s_1 \rightarrow m_1 \rightarrow s, \ldots, s_n \rightarrow m_n \rightarrow s$. When executing *backtrack_sc*, the local composition around $s$ contains at least $s_1(\ldots, s, \ldots), \ldots, s_n(\ldots, s, \ldots)$.

**Algorithm 6** *backtrack_sc*$(e)$

```
    % remove the current composition if it exists
    if get(current_sc, e) != NULL then
      STOP_current_composition(e)
    end if
 5: set_SC ← get(current_set_sc, e)
    if set_SC = ∅ then
      % s has no more solution with the selected mts
      mts ← backtrack_mts(e)
      set_SC ← possible_sc(mts, e)
10:   put(current_set_sc, e, set_SC)
      sc ← backtrack_sc(e)
    else
      %  s selects a solution sc w.r.t. some criteria
      sc ← select_SC(set_SC)
15:   % the service composition sc is connected
      connect(sc)
      put(current_set_sc, e, set_SC \ {sc})
      put(current_sc, e, sc)
    end if
20: return(sc)
```

### 5.6.4 Selecting service composition and MTS

We do not detail the two selecting functions used above (*select_MTS* and *select_sc*). Selecting a MTS is a process based on composition structure (more or less, selecting a pattern): the connector and the number of services involved. Some criteria could thus be the number of involved services or the type of connector. Selecting a composition focuses on services instances. A criterion could thus be:

- preferences of services, e.g., a service prefers to work with some "friends"; this is not a constraint as *no_link*, but just a preference and it does not influence the computation of possible compositions (algorithm *possible_sc*). Note that this kind of preferences could be used as some soft constraints considering a solver that achieves optimization, the objective function being to maximize the preferences of the services; the hard constraint (the constraints that must be satisfied) being the constraints we consider here (such as *no_link*).
- or re-use of services: after a backtrack, one could try to carry on with approximately the same services in order to minimize the number of new connection or shared information: the algorithms (especially the one related to stops or backtracks) could be refined in order to keep the work already done by some "employees".

### 5.6.5 Managing backtrack request

The *treat_bt_request* algorithm is executed when a service $s$ receives a backtrack request from a service $s'$, meaning that $s'$ (or a further composition built by $s'$)

was not able to execute a task (contained in the message $fup$ that $s$ had sent to $s'$). Using the *get* command indexed by the message $fup$ that caused the failure, $s$ retrieves the event $e$ it was treating and caused the follow-up message to $s'$. Then, $s$ performs a change of composition (*backtrack_sc*) w.r.t. the event $e$.

When executing *treat_bt_request*, the local composition around $s$ contains at least $s_1(\ldots,s,\ldots),\ldots,s_n(\ldots,s,\ldots),s(\ldots,s',\ldots)$ where

- $s_1$, ..., $s_n$ are the services that sent messages which composed the event $e$ for $s$
- $s \rightarrow fup \rightarrow s'$ is the follow-up message that $s$ sent to $s'$ to treat the event $e$ with the pattern $s(\ldots,s',\ldots)$.

**Algorithm 7** *treat_bt_request* **triggering event:** $s' \rightarrow bt\_request(s \rightarrow fup \rightarrow s') \rightarrow s$

$e \leftarrow get(event\_implies\_fup, s \rightarrow fup \rightarrow s')$
$backtrack\_sc(e)$

**Remark 3 (Avoiding some backtracks)** *Using some more elaborated constructs, the above algorithm could be refined in order to avoid some backtracks. For example, consider a $m\_out\_of\_n(s,s_1,\ldots,s_n)$ construct, meaning that $s$ sends a task to $n$ services, but expects only $m$ answers. When receiving a backtrack request from one of the $s_i$, $s$ could just "remove" this $s_i$ (or just stop considering it) from the pattern. Backtrack would occur only when the number of remaining services $s_i$ becomes less than $m$.*

## 5.7 Forwarding and informing when stopping treating an event

### 5.7.1 Forwarding stopping treating an event

The *STOP_current_composition* algorithm forwards a stop request to all the employees $s_i$ that $s$ called to treat $e$ (they will receive a *st_request*). The pattern $s(\ldots,s_i,\ldots)$ that $s$ had built to treat $e$ will disappear. Finally, $clean(e)$ cleans the memories that are related to the event $e$ (e.g., *current_set_mts*, *current_mts*, ...). Stopping the current composition plus cleaning the memory consists in fact in stopping treating the event $e$.

When entering this algorithm, the composition around $s$ contains at least the following patterns $s(s_1,\ldots,s_n)$ that $s$ built to treat the event $e$. After the algorithm, this pattern vanishes.

**Algorithm 8** *STOP_current_composition*$(e)$

$FUPS \leftarrow get(fups\_implied\_by\_e, e)$
**for all** $s \rightarrow fup_i \rightarrow s_i \ \in FUPS$ **do**
  $send(s \rightarrow \ st\_request(s \rightarrow fup_i \rightarrow s_i) \ \rightarrow s_i)$
**end for**
$s(s_1, \ldots, s_n) \leftarrow get(current\_sc, e)$
$remove(s(s_1, \ldots, s_n))$
$clean(e)$

### 5.7.2 Receiving a stop request

The *treat_st_request* algorithm is executed when a service $s$ receives an event *st_request* from $s'$, one of its employers for $e$: $s'$ had sent a message to $s$ ($s' \rightarrow fup \rightarrow s$) which is part of the event $e$ for $s$; but $s'$ stopped treating the event $e'$ for which it sent $s' \rightarrow fup \rightarrow s$. Thus, $s$ has also to stop treating the event $e$ which was implied by $e'$, since the message $s' \rightarrow fup \rightarrow s$ participating in $e$ vanishes.

$s$ looks for the event $e$ in which the message $s' \rightarrow fup \rightarrow s$ was used. All the messages composing $e$ (except $s' \rightarrow fup \rightarrow s$) are raised again (remember that they had been consumed when raising the event in which they participate); thus, they will be treated again by *treat_msg* to participate in another event for $s$. Hence, the work and the message of the employers of $s$ (except $s'$) are not lost.

Finally, $s$ then stops treating the event $e$: the employees of $s$ for $e$ are requested to stop.

When receiving such a request, the composition around $s$ contains at least the following patterns $s'(\ldots, s, \ldots), s'_1(\ldots, s, \ldots), \ldots, s'_n(\ldots, s, \ldots)$, where

- $s'(\ldots, s, \ldots)$ is the pattern in which we find the employer $s'$ sending the stop request to $s$
- $s'_1(\ldots, s, \ldots), \ldots, s'_n(\ldots, s, \ldots)$ together with $s'(\ldots, s, \ldots)$ made the event $e$ for $s$; $s'$ is already stopped.

**Algorithm 9** *treat_st_request* **triggering event:** $s' \rightarrow \ st\_request(s' \rightarrow fup \rightarrow s) \ \rightarrow s)$

% $s$ looks for $e$
$SE \leftarrow get(current\_treated\_events)$
$\exists e \in SE$ such that $s' \rightarrow fup \rightarrow s \ \in e$
**for all** $s'_i \rightarrow m_i \rightarrow s \ \in \ e \setminus \{s' \rightarrow fup \rightarrow s\}$ **do**
  $raise(s'_i \rightarrow m_i \rightarrow s)$
**end for**
$STOP\_current\_composition(e)$

# 6 The solving process

Constraint problems are basically represented by a set of decision variables and a set of constraints among these variables. The purpose of a solving process is therefore to assign a value to each variable such that the constraints are satisfied.

## 6.1 Complete solver vs. uncomplete solver

Solving a constraint problem involves many different techniques issued from different scientific communities: computer science, operational research or applied mathematics. Therefore, the principles and purposes of the proposed solving methods are very diverse. But, we may classify these methods in two main families, which differ on a fundamental aspect: complete methods whose purpose is to provide the whole set of solutions and incomplete methods which aim at finding one solution. On the one hand, using an exhaustive exploration of the search space, complete methods are able to demonstrate that a given problem is not satisfiable while incomplete methods will be ineffective in that case. On the other hand, incomplete methods, which explore only some parts of the search space with respect to specific heuristics, are often more efficient to obtain a solution and, moreover, for large instances with huge search space they appear as the only usable methods since complete methods become intractable.

## 6.2 Distributed solving process

Distributed Constraint problems arise when pieces of information about variables, constraints or both are relevant to independent but communicating agents. This is well suited for a diverse range of distributed real world problems (e.g., auctions, problems that require privacy, problems that are naturally distributed, ...) emerging from the evolution of computation and communication technologies. Dealing with resource restrictions (e.g., limits on time and communication), privacy requirements, cooperation, and conflict resolution strategies are some of the challenges for distributed constraint solving.

As for standard solving process, distributed solving process may be complete or incomplete.

## 6.3 Our distributed solving process

In our framework, the solving process is distributed, based on some local partial solutions of the global problem which is to build a service composition for achieving a task (e.g., a portal for printing pictures). We have local (without monitoring or broadcast system) asynchronous communication among the services.

In contrast with most of the distributing solving processes, the aim of our framework is to build and use local and partial solutions of the global problem as soon as they have been computed: although they can be backtracked, local solutions (i.e., pieces of service composition) are immediately built and used as composition. Moreover, some constraints dynamically appear or vanish as side effects of service execution, construction or destruction of some parts of composition, and inside messages. The dynamic feature of our framework implies that the notion of a global problem does not exist:

- the global problem is unknown a priori since we do not know what will be the computations of the solver and the content of the messages, and thus, the constraints that will be raised as side effects of these processes;
- for the same reason, the global problem remains unknown during execution and solving;
- even a posteriori we cannot know what was the global problem:
  - in case the global composition was built successfully and executed completely: some branches of the problem (i.e., some possible service compositions computed locally) were not tried, and thus, we cannot know what would have been all the constraints generated by these branches. However, in this case, the composition that was built and executed completely is a solution of the global problem (the parts of the global problem that are unknown would have been disjunctions or other branches of the global problem).
  - in case the composition cannot be completely built or executed: even in this case, the global problem is unknown. One may think that all the branches of the problem are explored in this case, but this is not the case for two reasons. First, we do not backtrack over events when several events could be raised; this issue could be easily overcome by a backtracking mechanism over events, or imposing restrictions over events as described later. Second, partial solutions are not kept: a local solution/composition is immediately built; if it fails, it vanishes whereas combining it with another local solution would have been successful. Formally, this issue could be overcome: it would require 1) keeping (as already built compositions) or memorizing (in the service which computed them) local solutions, 2) an heavy mechanism to make the service cooperate and synchronize in order to try all the combinations of these local solutions, 3) some "memorization" of the internal operations of services. In practice, this is impossible for several reasons. It would increase the number of messages and a tricky mechanism would be needed for tagging messages so that they can be propagated in the composition to synchronize distant services belonging to compatible parts of composition. Moreover, a "physical" operation (even one that raises constraints) may not be something that can be memorized (e.g., sending pictures).

## 6.4 Required local solvers

For the flexibility of our framework we dot not fix the types of constraints and the constraint language. Hence, the local solver related to each service cannot be defined precisely here. However, we give some hints with respect to some common classes of constraints that could be used.

A lot of the constraints we refered before can be treated as some linear arithmetic constraints. For example, constraints over version of services such as $l.p.version > d.p.version$, are linear inequations; constraints about quantitative quality of services would be linear equations. This types of constraint can be solved by numerous solvers such as solver based on domain reduction combined

with enumeration (see [4] for example). Solvers such as Gecode [26], the finite domain solver of ECLiPse [28] or of SWI-Prolog [29] are such examples.

Constraints such as $link(a, b)$ can be easily specified and solved by unification with a Prolog-like system.

Constraints such as $L.p \subseteq lab.p \wedge L.m \subseteq lab.m \wedge L.o \subseteq lab.o \wedge L.t \subseteq lab.t$ meaning $L$ must be of type $lab$ can be solved either by a set solver, unification, or finite domain solvers ([4]) considering the initial set of services is known.

More exotic constraints about quality or policy of services could be treated as user-defined constraints in systems such as ECLiPse, or could be handled and manipulated by systems such as the CHR [13].

If one is interested in having different types of constraints, and to leave open the framework with the possibility of new constraints, we think that a system such as CHR embedded in a Prolog system, e.g. ECLiPse or SWI-Prolog, is a very good candidate for implementing the *solve* function we use in the algorithm *possible_sc* of Section 5.5.

## 6.5 On the completeness of the solving process

Since we cannot even know the global problem, it is difficult to talk about completeness of the global complete solver: we may fail to find a solution if there is one. Indeed, services do not exclusively exchange solutions as it is generally the case for distributed solving, and local solutions are not completely propagated.

- Consider a service $s$ that connects to several employees $s_i$. Then, the $s_i$ do not know each others and have no connection, except through $s$ that may vanish in case of failure.
- Consider a service $s$ that has $n$ employers $s_i$ which participate in the event $e$ for $s$. This could be seen as a constraint since all the $s_i$'s are required even there is no direct connection between the $s_i$'s in the composition. Thus, if $s$ fails to complete a task given by the $s_i$'s, all of them will be required to backtrack, whilst $s$ could have treated successfully another event involving some of the $s_i$'s. But since the messages have been consumed and there is no backtrack over events, it is not possible anymore to treat this case.

Therefore, we cannot really talk about completeness as used in the constraint solving community since a composition can fail for other reasons than just not being able to build a composition due for example to constraint messages, overlap of events, etc. However, let us try to give some results on what could be understood as completeness for our framework, i.e., building a complete composition as specified by a set of MTS. By restricting events to one message, we can obtain the following result.

**Proposition 1 (Completeness of the solving process)** *Consider that all the local solvers from the services participating in the composition are complete. Then, if all the events raised by the services are composed of a single message and that a message can participate in only one event, then the global solving process is complete.*

At that point, a natural question arises: is it possible to get a "complete" solving process when considering complex events?

- Assume that events do not overlap, i.e., they are totally disjoint. Thus, a message can participate in only one event. This fixes the problem of not backtracking over events. This can be done by either checking that events are disjoint or by imposing an extra parameter in messages to specify in which event they participate.

  However, there remains an issue. Consider two services sending the same message. Both messages can be used in the same complex event (if the events are made of one message, this problem vanishes). Either this is a modeling error of the designer of the composition, or the designer wants to put the two services in concurrency: the first that sends the message will be chosen. In this case, it would be possible to obtain a kind of completeness modulo the non-determinism of concurrent services, or this case must be prohibited.

- one may consider to backtrack over events by raising again consumed messages in case of failure of a service. But in this case, termination is lost: the service may wait forever to raise an event it can achieve whereas asking its employers to backtrack would have solved the problem. In fact, this technique would consist in always accepting a local solution and never backtracking over a local solution. Thus, this would not lead to a complete solver.

# 7 Simulating control constructs of OWL-S

In this section, we show how our single construct $s(s_1, \ldots, s_n)$ and some MTS can be used to design a composition specified with OWL-S control constructs.

Our constructs seem very simple compared to the connectors introduced in OWL-S [18], which involve some more complex communications (e.g., question/answer) and protocols (e.g., concurrency).

However, with our single construct and the MTS, we can mimic the control constructs of OWL-S. We can easily support the following control construct of OWL-S; for each of them we describe the form of the composition and the skeleton of the various MTS. Note that this process could be automated to directly convert OWL-S code to our framework.

In the following, the *context* of a control construct corresponds to the other constructs occurring in the composition. For example, if a service $s$ must build a sequence $seq(s_1, s_2)$ and if $s_2$ must build a split $split(s_3, s_4, s_5)$, then $split(s_3, s_4, s_5)$ is in the context of $seq(s_1, s_2)$ (and vice-versa).

## 7.1 Sequence

In a $seq(s_1, s_2)$ construct, a service $s$ requests an operation $op_1$ to a service $s_1$, and uses the result of the operation for requesting an other operation $op_2$ to $s_2$. With the composition we propose below, this control construct corresponds to 4 constructs $s(s_1)$, $s_1(s)$, $s(s_2)$, and $s_2(s)$ and 5 MTS in our framework (3 for $s$, 1 for $s_1$ and 1 for $s_2$).

The composition starts when $s$ raises an event $e$ which requests to simulate and execute $seq(s_1, s_2)$. The MTS $mts_{seq1}$ associated to $e$ is related to one or

several employers of $s$, depending on the form of the event $e$ that triggered this construct.

The else part of the MTS is empty since $mts_{seq1}.switch = true$, and the constraint composition pattern $mts_{seq1}.then.CP$ is of the form $s(s_1)$ and the constraint $mts_{seq1}.then.C_{CP} = true$, i.e., the service $s_1$ is known, and thus there cannot be any constraint. $mts_{seq1}.then.fup$ is composed of one message $s \rightarrow m_1(D) \rightarrow s_1$ to request $s_1$ to execute operation $op_1$. The operation $mts_{seq1}.then.op$ can be empty (in this case $D$ is either fixed or included in the event $e$), or can participate in building the data $D$.

$s_1$ has been encapsulated with a MTS $mts_{s_1}$ in which *event* is $s \rightarrow m_1(\ldots) \rightarrow s_1$, $mts_{s_1}.select = 1$, $mts_{s_1}.switch = true$. The composition pattern only requires to connect to $s$ without any constraint: $mts_{s_1}.CP = s_1(s)$ and $mts_{s_1}.C_{CP} = true$. The operation is obviously $op_1$: $mts_{s_1}.op = op_1$. The follow-up message $mts_{s_1}.then.fup$ is an unique message $s_1 \rightarrow r_1(D_1) \rightarrow s$ in which data $D_1$ certainly contain some results of $op_1$.

In $s$, an event is raised when receiving a message $s_1 \rightarrow r_1(D_1) \rightarrow s$. The associated MTS $mts_{seq2}$ is as follows: $mts_{seq2}.select = 1$, $mts_{seq2}.switch = true$, $mts_{seq2}.CP = s(s_2)$, $mts_{seq2}.C_{CP} = true$, and $mts_{seq2}.then.fup$ is composed of one message $s \rightarrow m_2(D_1') \rightarrow s_2$ to request $s_2$ to execute the operation $op_2$; $D_1'$ maybe exactly $D_1$ (in this case $mts_{seq2}.then.op$ can be empty) or a modification of $D_1$ by $mts_{seq2}.then.op$.

$s_2$ has a MTS $mts_{s_2}$ similar to $mts_{s_1}$ but replacing $s_1$ by $s_2$, $op_1$ by $op_2$, ...

Finally, $s$ has a MTS $mts_{seq3}$ to receive the result of $op_2$ by $s_2$. We do not detail the rest of this MTS since it is based on the context of the $seq(s_1, s_2)$ construct in the framework of OWL-S.

Note that this is not the only way to simulate the *seq* construct. If one is not interested in the intermediate results from $s_1$, one can build a "linear" composition $s(s_1)$, $s_1(s_2)$, and $s_2(s)$ (to directly return the result to $s$) or $s_2(s_1)$, $s_1(s)$ (to return the result via $s_1$). In the first case, 4 MTS are required (2 for $s$, 1 for $s_1$, and 1 for $s_2$) while in the second case 5 MTS are required (2 for $s$, 2 for $s_1$, and 1 for $s$).

## 7.2 Split

In OWL-S, the components of a split process are a bag of services to be executed concurrently. Split completes as soon as all of its services have been scheduled for execution.

Consider a service $s$ that must simulate a $split(s_1, \ldots, s_n)$ in our framework. The composition starts when $s$ raises an event $e$ which requests to simulate and execute $split(s_1, \ldots, s_n)$. The MTS $mts_{split}$ associated to $e$ is related to one or several employers of $s$, depending on the form of the event $e$ that triggered this construct (e.g., depending on the context of the Split).

The MTS associated to $e$ is rather simple. The constraint composition pattern $mts_{split}.CP$ is of the form $s(s_1, \ldots, s_n)$ and the constraint $mts_{seq1}.C_{CP} = true$, i.e., the services $s_i$ are given, and thus there cannot be any constraint. $mts_{seq1}.switch = true$ and $mts_{seq1}.then.fup$ is composed of $n$ messages $s \rightarrow m(D) \rightarrow s_i$ to request each of the $s_i$ to execute an operation associated with $m$. The operation $mts_{split}.op$ can be empty (in this case $D$ is either fixed or included in the event $e$), or can participate in building the data $D$.

Each of the $s_i$ has at its disposal the MTS $mts_{splitted}$ in which *event* is $s \rightarrow m(D) \rightarrow s_i$, $mts_{splitted}.select = 1$, $mts_{splitted}.switch = true$. The composition pattern and the follow-up messages depends on the context of the Split and the operation is $op_1$.

Note that here we have 1 construct and 2 MTS since we consider the same operation and the same composition around each $s_i$. One could consider different operations and/or compositions: in this case, one should consider one MTS per $s_i$, and $s$ should send specific follow-up messages (i.e., $s \rightarrow m_i(D_i) \rightarrow s_i$).

## 7.3 Split+Join

This construct consists of concurrent execution of a set of services with barrier synchronization: split+join completes when all the $s_i$ have completed.

Consider a service $s$ that must simulate a $split + join(s_1, \ldots, s_n)$. Then $s$ has a MTS $mts_{split+join}$ identical to $mts_{split}$.

The $s_i$ will use the MTS $mts_{s+j}$ which is derived from $mts_{splitted}$ by imposing that $mts_{s+j}.CP$ contains $s_i(s)$ and that $mts_{s+j}.fup$ contains a message $s_i \rightarrow m(D_i) \rightarrow s$.

It remains treating the answer of the $s_i$. To this end, $s$ will raise an event $e = s_1 \rightarrow m(D_1) \rightarrow s, \ldots, s_n \rightarrow m(D_n) \rightarrow s$. The MTS $mts_{split+join2}$ associated to $e$ is as follows: $mts_{split+join2}.select = all$ to retain all the messages; the rest depends on the context of the split+join.

## 7.4 Choice

Choice is similar to split+join. The only modification is to change the selection and selection criteria of MTS $mts_{split+join2}$: *select* must be set to 1, and the selection criteria must be set to *indeterminism* (whatever response of the $s_i$), or $first$ (the fastest one), or a best fit depending on an evaluation of each of the answer.

## 7.5 If-Then-Else

Consider a service $s$ that must simulate an $if\_then\_else(s_1, s_2)$. Then $s$ has a MTS $mts_{ite}$ such that the *switch* is not empty. Depending on the evaluation of the switch, the else part of the then part of the MTS is executed. We have that $mts_{ite}.then.CP = s(s_1)$ and $mts_{ite}.else.CP = s(s_2)$. The operations of each part can be different; they depend on the context of the $if\_then\_else$. The follow-up of the then part is a message to $s_1$ $(s \rightarrow m_{then}(D_{then}) \rightarrow s_1))$ whereas the else part has a message to $s_2$ $(s \rightarrow m_{else}(D_{else}) \rightarrow s_2))$; both data and message names can be different.

## 7.6 Any-Order

This control construct allows the services to be executed in some unspecified order but not concurrently. Execution and completion of all components is required.

In our framework, this construct can be simulated with several *seq* control constructs $seq(s_1, s_2)$, $seq(s_3, s_4)$, ..., and terminating with a simple construct $s(s_n)$ (if $n$ is odd) or $seq(s_{n-1}, s_n)$. To chain the *seq* constructs, the MTS

$mts_{seq3}$ (which was not completely defined above because we did not know the context) can now be completed similarly to $mts_{seq1}$ (recall that this MTS initiate a *seq* construct).

### 7.7 Iterate

The iterate construct makes no assumption about how many iterations are made. The service $s$ starting an $iterate(s_1)$ control construct has thus a MTS $mts_{it}$ such that: the event $e$ depends on how many employers $s$ has, $mts_{it}.switch = true$, $mts_{it}.then.CP = s(s_1)$, $mts_{it}.C_{CP} = true$, $mts_{it}.then.fup = s \to m \to s_1$.

$s_1$, with the event $s \to m \to s_1$ will match the MTS $mts_{iterate}$, in which the switch is set to true, and $mts_{iterate}.then.fup = s_1 \to m' \to s$.

$s$ has a second MTS $mts_{it2}$, which is similar to $mts_{it}$, with the event being $s_1 \to m' \to s$. Note that if $s$ has only one employer at the beginning, then $mts_{it}$ can be $mts_{it2}$.

### 7.8 Repeat

Repeat-While and Repeat-Until both are iterates until a condition becomes false or true. These two constructs can be simulated by modifying the simulation of the iterate construct: the *switch* of $mts_{it}$ and $mts_{it2}$ is not empty. If the condition is about the data of the messages, nothing special has to be added. If the condition is about a loop counter, then this counter can be added to the data of messages $m$ and $m'$: $s$ will increment or decrement this counter at each loop and the switch will be of the form $counter = n$, or $counter < n$, ... The else part of these MTS is not empty and depends on the context.

Note that if one does not want to add a new parameter (the counter) to the message, one can keep the counter inside $s$: the MTS must be completed such that $s$ sends itself a message; the event is then composed of the message from $s_1$, and the message from $s$ to $s$. When the switch becomes false, $s$ does not send itself the message, and thus the event for carrying on will not be raised.

## 8 Related work

Web service composition is nowadays a very active research direction. Many approaches have been investigated including techniques based on planning in AI [23, 27], situation calculus [21, 19, 10], conversational transition systems [11], or symbolic model-checking applied to planning [24, 25]. Our model relies on the use of constraint (logic) programming. Applying extensions of logic programming to Web service composition has been already investigated. The seminal approach presented in [19] shows how an extension of Golog (implementing a situation calculus) provides a well-suited formalism for the composition problem.

The fundamental issue addressed by all these approaches consists in automatically building a composition schema that fulfills a client request via a combination of existing services. The automata-based approach is currently very popular. In [11], Web services exchange asynchronous messages and they are modeled as Mealy machines, but there is no way to handle data, for instance

as parameters of messages. In [7, 5], Web service are represented deterministic transition systems involving operations and exchanges of messages. In [6], the authors present the Colombo framework in which automata incorporate conversational aspects, parametrized operations and updates in a common database. In this framework, a mediator is built to perform the interaction between services needed to achieve a given goal, expressed as a goal service.

In our approach, an abstract form of the composition is given by using patterns that contain typed variables to represent different types of services. Depending on constraints between services and the constraints carried by the exchanged messages, these variables still have to be instantiated, to obtain an executable composition involving concrete services.

There are few papers reporting experiments on the use of constraint reasoning for the composition problem [15, 12, 2, 3, 17, 16]. We have already discussed some preliminary ideas in [20]. As in [15], we do not consider all the dimensions of the problem, since we assume that a pattern composition is already known, and we restrict us to the problem of instantiating the variables of the pattern, i.e. the different kinds of Web services. Contrary to [15], we do not consider constraints globally, but we handle constraints locally in order to build a solution gradually using a top-down mechanism. Our framework has the ability to tackle privacy requirements (as opposed to [15]) by considering only public properties of possible sub-services, by limiting exchange of data, and by keeping locally most of the knowledge. In [2], the authors use an integer programming solver by assuming that constraints and objective functions are linear. In [3], the idea is to consider the composition problem as particular constraint-based configuration problem. In that paper, a goal is expressed by a set of output messages expected by the composition. The configurator used in this context is goal-oriented and apparently proceeds by applying backward chaining techniques.

Due to similarities between constraint reasoning and AI planning [22], our approach could be compared to the use of planning systems for building a solution (a plan) to a composition problem. The Hierarchical Task Network (HTN) planning seems particularly well-suited to handle the (Hierarchy of) services involved in a pattern to be instantiated [27]. As shown in [27], the HTN planning system SHOP2 can be applied to build a solution to the composition problem. However, the planning process presented in that paper is restricted by the capabilities of the planning system, which cannot handle the concurrency.

# 9 Conclusion

In this paper, we promote the use of constraint reasoning to implement a form of pattern instantiation. With respect to classical configuration problems, our approach has to cope with the dynamic aspect of Web services. We have proposed a framework where Web services interact via queries and answers, which are respectively the input and output messages exchanged by Web services. Hence, our approach relies on the analogy between the computation performed by a Web service and the execution of a constraint (logic) program. In order to take into account the dynamic behavior of Web services, the facts needed to execute the constraint reasoning engine of each Web service are not static, but are obtained dynamically by calling sub-services. In our framework, we consider

time-dependent costs, and temporal constraints imposed by the client, which are analyzed with respect to estimated durations indicated by sub-services. Moreover, the process is monitored, to possibly change on-the-fly the current selection of Web services. This monitoring phase is also supported by the constraint engine associated to the Web service.

# References

[1] Web Services Business Process Execution Language. http://www.oasis-open.org.

[2] Rohit Aggarwal, Kunal Verma, John A. Miller, and William Milnor. Constraint driven web service composition in meteor-s. In *Proc. of SCC*, pages 23–30. IEEE, 2004.

[3] Patrick Albert, Laurent Henocque, and Mathias Kleiner. Configuration-Based Workflow Composition. In *2005 IEEE International Conference on Web Services (ICWS 2005), 11-15 July 2005, Orlando, FL, USA*, pages 285–292. IEEE Computer Society, 2005.

[4] K. R. Apt. *Principles of Constraint Programming*. Cambridge Univ. Press, 2003.

[5] Daniela Berardi. *Automatic Service Composition. Models, techniques and tools*. PhD thesis, La Sapienza University, Roma, 2005.

[6] Daniela Berardi, Diego Calvanese, Giuseppe De Giacomo, Richard Hull, and Massimo Mecella. Automatic composition of transition-based semantic web services with messaging. In *Proc. of VLDB*, pages 613–624. ACM, 2005.

[7] Daniela Berardi, Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Massimo Mecella. Automatic composition of e-services that export their behavior. In *Proc. of ICSOC*, volume 2910 of *LNCS*, pages 43–58. Springer, 2003.

[8] Daniela Berardi, Diego Calvanese, Giuseppe De Giacomo, and Massimo Mecella. Composition of services with nondeterministic observable behavior. In *Proceedings of Third International Conference on Service-Oriented Computing - ICSOC 2005, Amsterdam, The Netherlands*, volume 3826 of *LNCS*, pages 520–526, 2005.

[9] Sami Bhiri, Olivier Perrin, and Claude Godart. Ensuring required failure atomicity of composite web services. In *Proceedings of the 14th international conference on World Wide Web, WWW 2005, Chiba, Japan, May 10-14, 2005*, pages 138–147, 2005.

[10] M. Bienvenu, C. Fritz, and S. McIlraith. Planning with qualitative temporal preferences. In *Proc. of KR*, pages 134–144, 2006.

[11] Tevfik Bultan, Xiang Fu, Richard Hull, and Jianwen Su. Conversation specification: a new approach to design and analysis of e-service composition. In *Proc. of WWW*, 2003.

[12] Nizamuddin Channa, Shanping Li, Abdul Wasim Shaikh, and Xiangjun Fu. Constraint satisfaction in dynamic web service composition. In *Proc. of DEXA*, pages 658–664. IEEE, 2005.

[13] T. Fruehwirth. *Constraint Handling Rules*. Cambridge University Press, 2009.

[14] Nawal Guermouche, Olivier Perrin, and Christophe Ringeissen. Timed specification for web services compatibility analysis. *Electr. Notes Theor. Comput. Sci.*, 200(3):155–170, 2008.

[15] Ahlem Ben Hassine, Shigeo Matsubara, and Toru Ishida. A constraint-based approach to horizontal web service composition. In *Proc. of Semantic Web Conference*, volume 4273 of *LNCS*, pages 130–143. Springer, 2006.

[16] Srividya Kona, Ajay Bansal, and Gopal Gupta. Automatic Composition of Semantic Web Services. In *2007 IEEE International Conference on Web Services (ICWS 2007), July 9-13, Salt Lake City, Utah, USA*. IEEE Computer Society, 2007.

[17] Alexander Lazovik, Marco Aiello, and Rosella Gennari. Encoding requests to web service compositions as constraints. In *Proc. of CP*, volume 3709 of *LNCS*, 2005.

[18] David L. Martin, Mark H. Burstein, Drew V. McDermott, Sheila A. McIlraith, Massimo Paolucci, Katia P. Sycara, Deborah L. McGuinness, Evren Sirin, and Naveen Srinivasan. Bringing semantics to web services with owl-s. In *World Wide Web*, pages 243–277, 2007.

[19] Sheila A. McIlraith and Tran Cao Son. Adapting golog for composition of semantic web services. In *Proc. of KR*, pages 482–496. Morgan Kaufmann, 2002.

[20] Eric Monfroy, Olivier Perrin, and Christophe Ringeissen. Modeling Web services Composition with Constraints. In *Proc. of 3CCC — Colombian Conference on Computer Science*, 2008. To appear also in "Revista Avances en Sistemas e Informtica", Vol. 5, Nr. 2.

[21] Srini Narayanan and Sheila A. McIlraith. Simulation, verification and automated composition of web services. In *Proc. of WWW*, pages 77–88, 2002.

[22] Alexander Nareyek, Eugene C. Freuder, Robert Fourer, Enrico Giunchiglia, Robert P. Goldman, Henry A. Kautz, Jussi Rintanen, and Austin Tate. Constraints and AI Planning. *IEEE Intelligent Systems*, 20(2):62–72, 2005.

[23] Joachim Peer. Web Service Composition as AI Planning - a Survey. Technical Report Univ. of St. Gallen, 2005.

[24] Marco Pistore, Fabio Barbon, Piergiorgio Bertoli, Dmitry Shaparau, and Paolo Traverso. Planning and Monitoring Web Service Composition. In *Artificial Intelligence: Methodology, Systems, and Applications, 11th International Conference, AIMSA 2004, Varna, Bulgaria, September 2-4, 2004, Proceedings*, volume 3192 of *Lecture Notes in Computer Science*, pages 106–115. Springer, 2004.

[25] Marco Pistore, Annapaola Marconi, Piergiorgio Bertoli, and Paolo Traverso. Automated composition of web services by planning at the knowledge level. In *IJCAI*, pages 1252–1259, 2005.

[26] Christian Schulte and Peter J. Stuckey. Efficient constraint propagation engines. *Transactions on Programming Languages and Systems*, 31(1):2:1–2:43, December 2008.

[27] Evren Sirin, Bijan Parsia, Dan Wu, James A. Hendler, and Dana S. Nau. HTN planning for web service composition using shop2. *J. Web Sem.*, 1(4):377–396, 2004.

[28] Mark Wallace, Stefano Novello, and Joachim Schimpf. ECLiPSe: A Platform for Constraint Logic Programming. *ICL Systems Journal*, 12(1):159–200, 1997. Revised version of Technical Report IC-Parc, Imperial College, London, August 1997.

[29] Jan Wielemaker, Zhisheng Huang, and Lourens van der Meij. Swi-prolog and the web. *Theory and Practice of Logic Programming*, 8(3):363–392, 2008.

Centre de recherche INRIA Nancy – Grand Est
LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)

Centre de recherche INRIA Bordeaux – Sud Ouest : Domaine Universitaire - 351, cours de la Libération - 33405 Talence Cedex
Centre de recherche INRIA Grenoble – Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier
Centre de recherche INRIA Lille – Nord Europe : Parc Scientifique de la Haute Borne - 40, avenue Halley - 59650 Villeneuve d'Ascq
Centre de recherche INRIA Paris – Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex
Centre de recherche INRIA Rennes – Bretagne Atlantique : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex
Centre de recherche INRIA Saclay – Île-de-France : Parc Orsay Université - ZAC des Vignes : 4, rue Jacques Monod - 91893 Orsay Cedex
Centre de recherche INRIA Sophia Antipolis – Méditerranée : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399